Theoretical Computer Science 26 (1983) 343-353
North-Holland

NOTE

COMPUTING THE BEHAVIOUR OF ASYNCHRONOUS PROCESSES

John STAPLES and V.L. NGUYEN*
Department of Computer Science, University of Queensland, St. Lucia, Queensland 4067, Australia

Communicated by Z. Manna
Received May 1982
Revised January 1983

Abstract. We describe a variation of Brock-Ackerman's (1981) model of nondeterministic asynchronous process, and devise a computational method for characterizing asynchronous processes defined by networks. This method is shown to be equivalent to a method of combining traces related to that of Brock and Ackerman. Recursively defined networks are also considered.

1. Introduction

Much attention has been given recently to distributed computation by networks of processes, where the processes communicate solely by the asynchronous transmission of messages through input and output ports. This form of computation is called data flow, for processes can function concurrently as long as data are available; there is no control flow. Processes may exhibit memory behaviour. That is, the current output may depend on past inputs as well as the current inputs. There is no shared memory between processes in a network, hence data flow computation has locality of effect.

In order to describe the semantics of such processes we seek a model which is abstract, that is, which can hide the internal states of the processes and can characterize processes solely through their input–output behaviour. The model must also provide methods to describe processes defined by networks in terms of the behaviour of the networks' components. At least some of these methods must be computational, rather than descriptive, in nature.

In this paper we give a variation of Brock–Ackerman's model of processes and devise a computational method for describing the behaviour of networks of processes. Our method generalizes the successful method of Khan [3] for functional

* Present address: Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.

(which Kahn called deterministic) processes. A previous proposal in this direction has been made by Kosinski [5]; we indicate an error in that paper.

Several previous proposals have been made for characterization of asynchronous networks by combining traces of the networks' components: for example, Brock and Ackerman [2] and Pratt [7, 8]. These are descriptive rather than computational; they do not provide a method for computing nondeterministic network behaviour by approximation, as does Kahn's fixed-point method for functional processes.

We also compare our approach with a recent contribution by Back and Mannila [1] to the theory of synchronous processes.

2. Background

2.1. Kahn's model

In Kahn's model [3] a functional process is characterized as a set of functions mapping tuples of input sequences to output sequences. It is simple and elegant. Because fixpoint semantics applies to it, a network of processes can be characterized as a system of recursive equations. Its behaviour for given input sequences can be obtained, and approximations to its behaviour computed, by the fixpoint method. Unfortunately, however, it does not straightforwardly extend to nonfunctional processes, as shown in [2].

Recently, several authors have proposed models of networks of general asynchronous processes. We now briefly describe three of these models, which have influenced our model directly or indirectly.

2.2. Brock–Ackerman's model

In this model [2] a process is defined to be a set of 'scenarios'. Each scenario is a partially ordered multiset of input and output events, where an event is an item of data associated with an input or output port. Intuitively, one event is less than another if the former must precede, or cause the latter.

There is no causality relation between events at different input ports, between events at different output ports, or from output events to input events. In any scenario, the events at a single port are totally ordered. The following notation is used. For a given port of a given scenario, the sequence of data items defined by the increasing sequence of events at that port is called the *history* of that port in that scenario.

A network is a scheme for combining processes by making pairwise identifications of some output ports with some input ports. Nominating processes for each of the nodes in the network should define a new composite process. In the case of the Brock–Ackerman model the composite process is defined as follows. We consider this definition as an example of a 'trace combining method' for the definition of composite processes.

(i) Form the Cartesian product of the processes.

(ii) Select from it each tuple of scenarios such that at each pair of identified ports the histories are the same.

(iii) Merge each such tuple into one 'network scenario', by pairwise identification of the events at identified ports.

(iv) Discard network scenarios which contain directed cycles.

(v) Derive a set of (process) scenarios from the remaining network scenarios by deleting all events other than events at input or output ports of the network. This set of scenarios characterizes the process defined by the network.

2.3. Pratt's model

This model [7, 8] also provides an example of the 'trace combining method'. Pratt defines a process to be a set of *traces*. Traces, like scenarios, are partially ordered multisets of events, but in this case:

(i) Only finite traces are considered.

(ii) The partial order is intended to represent temporal order, so that there is no further restriction on ordering of events.

(iii) In particular, events may extend in time, so that even events at a given port are not required to be totally ordered.

Composite processes are formed from networks of components as in Brock–Ackerman's model, except that (to accord with the time order motivation) at the stage corresponding to (iv) in the description of the Brock–Ackerman model, a network trace is discarded if it contains either a directed cycle or any augmentation of the original partial order on events of any component process.

Both of the above models are abstract, but are descriptive rather than computational in their characterization of composite processes. Our main aim in this note is to describe a computational approach. Before that, however, we review a previous attempt in that direction.

2.4. Kosinski's model

In this model [5] processes are defined to be functions which transform sets of tagged sequences of data to sets of tagged sequences of data. Each tag represents a nondeterministic choice which has been made in order to reach the state at which that data item was produced; so that this model does not succeed in giving an input–output description of the process. It is not abstract in the sense described above.

The method of [5] depends on applying conventional fixpoint theory to the model, for which purpose [5] proposes to define a chain-complete partial order on tagged-sequence sets. However, the example given in Appendix A shows that the argument in [5] for chain-completeness fails.

3. A variation of Brock–Ackerman's model

3.1. Definitions

As usual, a process is assumed to have a finite sequence of input ports, say $i_1, \ldots, i_m$, and a finite sequence of output ports, say $o_1, \ldots, o_n$. An *event* is an association of a data item to a port.

The main difference from Brock and Ackerman's model is that, whereas they consider sets of complete scenarios (or traces), we consider ideals of finite partial traces, as defined below.

A *trace* of a process is defined to be a finite partially ordered multiset of events (of that process) where events at the same port are assumed to be totally ordered. Intuitively, the order represents causality order. A trace represents a partial history of the input–output behaviour of the process.

A trace may be depicted graphically by columns of data, one column for each port. Each column represents the sequence of events at its port. A partial order is implied by the following:

(i) The assumption that each column is totally ordered.

(ii) Additional orderings which may be sketched by means of arrows from certain input events to certain output events.

(See for example Fig. 1.)

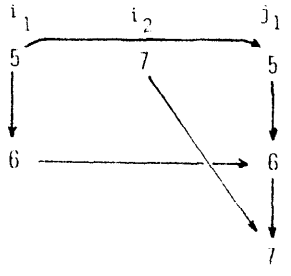

Fig. 1.

Since the data flow computation is asynchronous, events at different input ports, and events at different output ports, are incomparable; and no output event is less than an input event.

A convenient notation is that of an *ideal* I of a partial order P: $I \subseteq P$ is an ideal if $x \in I$, $y \in P$ and y implies $y \in I$. The ideal *generated by* $S \subseteq P$ is just

$$\{x \in P: x \leq s \text{ for some } s \in S\}$$

A *subtrace* of a trace is defined to be an ideal of that trace. We write $U \leq T$ to denote that the trace U is an ideal of the trace T, thereby defining a partial ordering of traces.

A process over a set of input and output ports is an ideal of the set of all traces over that port-set and contains all traces with empty outputs in this set. That is, if

a trace belongs to a process, so do all its subtraces. Hence, the traces of a process describe incomplete as well as complete behaviours of the process.

3.2. Networks of processes

In the network schemes we consider, there are places for finitely many processes, each with finitely many ports. Static connections are made from output ports to input ports. Each input port can be connected to at most one output port, and vice versa. Connections from output ports to input ports of the same process are permitted. The connected input and output ports are hidden. Fanout and hiding of disconnected input and output ports can be accomplished by use of appropriately defined processes, as in [7, 8].

To construct a process from component processes according to a net scheme in this way we use two kinds of operation on processes, as in [6] and as follows.

3.2.1. Disjoint union

To avoid irrelevant details about renaming ports, we shall assume that no two of the processes whose disjoint union is to be formed have any port names in common.

Given n such processes $P_1, \ldots, P_n$, their disjoint union $\bigcup(P_1, \ldots, P_n)$ is the process whose input ports and output ports are those of $P_1, \ldots, P_n$ and whose traces are disjoint unions of traces, one from each of $P_1, \ldots, P_n$ (see, for example, Fig. 2).

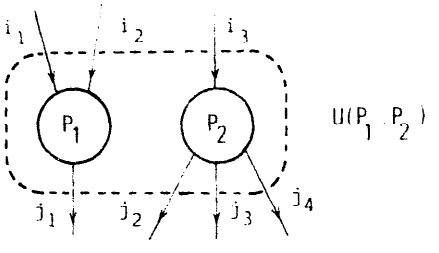

Fig. 2.

3.2.2. Linking

If P is a process with m input ports and n output ports, then let

$$L^{i_1,\ldots,i_k}_{j_1,\ldots,j_k}(P), \quad 1 \leq k \leq m,$$

where each i_q is a distinct input port name and each j_q is a distinct output port name, denote the process obtained from P by connecting j_q to i_q, $q = 1, \ldots, k$, then hiding all the input and output ports so connected (see Fig. 3).

More precisely, the required process is defined much as in the Brock–Ackerman approach sketched above, but with minor modifications to accommodate the more general networks considered here. In particular we allow connections from an output port to an input port of the same process. We proceed as follows:

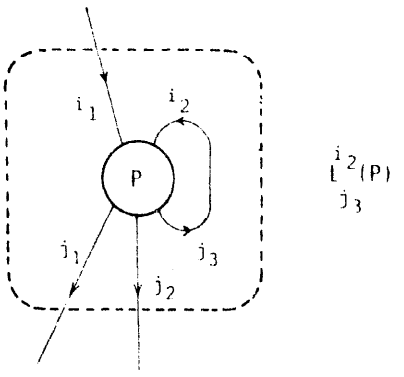

Fig. 3.

(i) Select from P the traces such that at each pair of linked ports the histories are the same.

(ii) Extend, if possible, the partial order of each trace by requiring that each event at a linked output port be less than the corresponding event at the input port to which it is linked. If no such extended partial order exists, discard the trace.

(iii) Delete from the result all events at linked input and output ports, to achieve a trace again.

The ideal of traces constructed in this way comprises the process

$$L^{i_1 \ldots i_k}_{j_1 \ldots j_k}(P).$$

3.3. *Associativity of network construction*

Our aim here is to show that it is immaterial whether the process defined by a network is constructed all at once, or stepwise by first constructing subnetworks.

As well as being a useful technical property, this associativity is necessary in order to model processes accurately. It rules out anomalies such as that of Brock and Ackerman. Checking that it holds is a basic test of the adequacy of our definitions.

It is necessary to show that linking and disjoint union commute, and that each of the operations of disjoint union and linking is associative, in the sense that they may be performed sequentially in any order or in parallel without changing the resulting process.

The first two are evident. We consider the third. It is enough to show that

$$L^a_b(L^c_d(P)) = L^{ac}_{bd}(P)$$

where a, b, c, d denote appropriate strings of port names.

The inclusion $L^a_b(L^c_d(P)) \supseteq L^{ac}_{bd}(P)$ is evident. Conversely, consider a trace T of $L^a_b(L^c_d(P))$, deriving say from a trace S of $L^c_d(P)$, which derives in turn from a trace R of P. If adding the links of L^{ac}_{bd} to R gives a cycle, then that cycle must involve links from $L^a_b(P)$, or else S has a cycle; a contradiction. But then it defines a cycle in T; a contradiction; so that R defines a trace in $L^{ac}_{bd}(P)$, which can only be T.

3.4. A computational method for describing processes defined by networks

Notation. If T is a trace and x is a port, denote by $T(x)$ the history of x in T, that is, the sequence of events at X in T.

If P is a process with input ports $i_1, \ldots, i_m$, denote by $P(a_1, \ldots, a_m)$ the set of all traces T in P such that $T(i_k) = a_k$, $k = 1, \ldots, m$.

We first describe how the traces of $L^{i_k}_{o_k}(P)$ can be obtained from those of P. Consider the computation of an arbitrary trace of $L^{i_k}_{o_k}(P)$ with given input histories $a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_m$ at $i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_m$ respectively. The correctness of this algorithm will be established in the following section.

Algorithm

Step 1. Choose $T_1 \in P(a_1, \ldots, a_{k-1}, \perp, a_{k+1}, \ldots, a_m)$.

Step $h+1$. Choose any $T_{h+1} \in P(a_1, \ldots, a_{k-1}, T_h(o_k), a_{k+1}, \ldots, a_m)$ such that $T_h \leqslant T_{h+1}$. Note that $T_{h+1}(i_k) = T_h(o_k)$.

This algorithm is terminated after an arbitrary, finite number of steps, but for convenience we represent it by an infinite, increasing but eventually constant sequence.

Observe that $T_1 < T_2 < \cdots < \text{lub } T_h$, and that

$$\text{lub } T_h(o_k) = \text{lub } T_h(i_k).$$

Now lub T_h is in P, since it equals T_h for some h. We extend the partial order of lub T_h by making the qth event at o_k less than the qth event at i_k, $q = 1, 2, \ldots$, then delete lub $T_h(i_k)$ and lub $T_h(o_k)$ from the extended trace to obtain a trace T. $L^{i_k}_{o_k}(P)(a_1, \ldots, a_m)$ includes just the traces obtained in this way, as is shown below.

Note that in this method we finitely compute all traces of the process without the need for identification of ports or consistency checks. Consistency is ensured by requiring $T_{h+1} \geqslant T_h$ and $T_{h+1}(i_k) = T_h(o_k)$.

The above iterative method straightforwardly extends to the computation of traces resulting from simultaneous linking of multiple pairs of ports.

3.5. Equivalence of the iterative method and the trace-combining method

We first consider a trace S which is generated by the iterative method and show that it is also provided by the trace-combining method. That is, S is obtained from some lub $T_h \in P$ by extending the partial order and then deleting the lub $T_h(i_e)$'s and lub $T_h(o_e)$'s.

Now, for any $T \in P$ which satisfies the condition $T(i_e) \leqslant T(o_e)$, $e = 1, \ldots, k$, a trace of $L^{i_1, \ldots, i_k}_{j_1, \ldots, j_k}(P)$ can be formed by extending the partial order so as to make linked input events *greater* than the corresponding output events (reflecting the intuition that the input event is caused by the prior output event), then deleting

the input and output events so linked—provided that the extended relation is a partial order.

In particular, T_1 satisfies these conditions, since $T_1(i_e) = \perp$, $e = 1, \ldots, k$. Also, suppose T_h satisfies these conditions. Then T_{h+1} does also, since the only new orderings added to T_{h+1} are from elements in T_{h+1} to elements in $T_{h+1} \backslash T_h$; that is, from output events to input events.

As $T_h = \bigcup_{h=1}^{\infty} T_h$, lub T_h also satisfies the conditions, and it is in P by assumption. Hence S can be obtained by the trace combining method.

Now suppose conversely that S is obtained by the trace combining method; say S results from a trace T of P, by extending its partial order by linking to give T^+, then deleting input and output ports.

Let T_1^+ be the largest ideal R of T^+ such that $R(i_e)$ is empty, $e = 1, \ldots, k$, and for $h > 0$ let T_{h+1}^+ be the largest ideal R of T^+ such that $R(i_e) = T_h^+(o_e)$, $e = 1, \ldots, k$.

Clearly $T_h^+ \leqslant T_{h+1}^+$ for all h, $\text{lub}_h T_h^+ = \bigcup_{h=1}^{\infty} T_h^+$, lub $T_h^+(i_e) = \text{lub}\, T_h^+(o_e)$, $e = 1, \ldots, k$ and lub $T_h^+ \leqslant T^+$.

To see that lub $T_h^+ = T^+$, suppose to the contrary that lub $T_h^+ < T^+$. Then there is a d such that lub $T_h^+(i_d) < T^+(i_d)$.

For otherwise, lub $T_h^+(i_e) = T^+(i_e)$, $e = 1, \ldots, k$, so as traces are finite, lub $T_h^+(o_e) = T^+(o_e)$, $e = 1, \ldots, k$; a contradiction.

This implies that there is an r such that

$$T_r^+(i_d) = T_h^+(i_d) \quad \text{for all } h \geqslant r.$$

Write a_1 for the first element of $T^+(i_d) \backslash T_r^+(i_d)$. Since a_1 is not in $T_r^+(i_d)$, a_1 is greater than some element, b_1 say, of some $T^+(o_e) \backslash T_r^+(o_e)$, and in view of the links we have added, b_1 is greater than some element a_2 of $T^+(i_e) \backslash T_r^+(i_e)$. Accordingly, $a_1 > a_2$.

The argument can be repeated indefinitely to generate an infinite strictly decreasing sequence $(a_n : n > 0)$ of elements of $\bigcup_{e=1,\ldots,k} (T^+(i_e) \backslash T_r^+(i_e))$; a contradiction since T^+ is finite.

Now define T_h to result from T_h^+ by restricting its partial order to be a subset of the partial order of T. Then lub $T_h = T$. It is clear that $\{T_h\}$ is a sequence defined by our iterative method. Accordingly, S can be produced by the iterative method, so that the two methods are equivalent.

4. Recursively defined networks

It is clear that processes defined by networks are also ideals of traces. Hence any network scheme can be regarded as a function from tuples of processes to processes.

Let $P_{o_1,\ldots,o_n}^{i_1,\ldots,i_m}$ denote the partially ordered set of all processes with given input ports $i_1, \ldots, i_m$ and output ports $o_1, \ldots, o_n$, where $P_1 \leqslant P_2$ means $P_1 \subseteq P_2$. Clearly

$P^{i_1,\ldots,i_m}_{o_1,\ldots,o_n}$ is a cpo. Its bottom element is the process consisting of all traces with empty outputs.

The operations we considered for forming processes as networks of component processes were defined elementwise. Consequently every network scheme defines a continuous function. The usual fixpoint method can now be applied to recursively defined networks.

The finite traces of a recursively defined process $P = \text{lub}\, P_n$ can be computed by an extension of the method given above. Each such trace is a trace of some P_n. Each P_n is the process defined by a finite, nonrecursive network obtained by substitution: that is, if P solves $P = f(P)$, then $P_n = f^n(\perp)$.

If, for example, P solves the equation depicted in Fig. 4, then the networks defining P_n's are as sketched in Fig. 5.

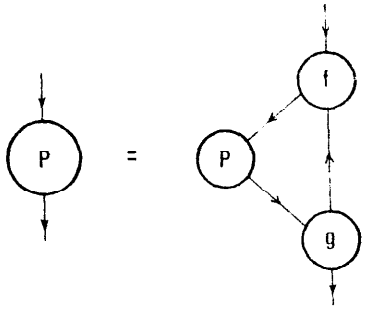

Fig. 4.

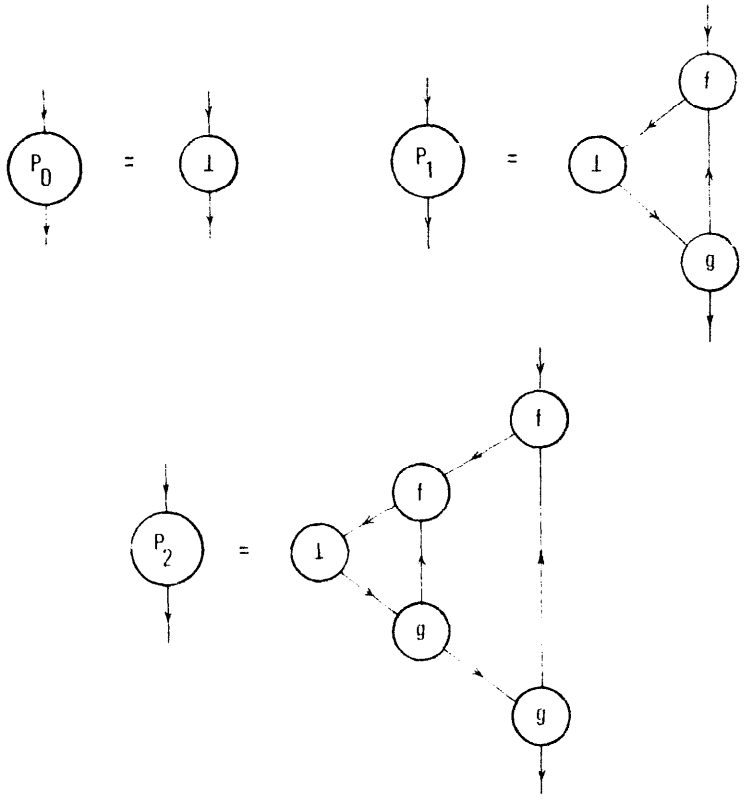

Fig. 5.

5. Comparison with the model of Back and Mannila

At a referee's request we compare the work of the current paper with the recent work [1] of Back and Mannila.

That work takes an approach similar to that of Pratt outlined above, but is less general in its concept of trace and is applied to synchronous processes. References to its antecedents in the theory of synchronous processes are given in [1]. Its main result is an abstraction theorem, which is the result in their theory corresponding to the associativity result we proved above. A corresponding result for a formalization of Pratt's approach [7] has been given in [6].

Back and Mannila propose a model of functional asynchronous processes in their system, in order to argue that their method can model Kahn's fixpoint method for such processes. That argument shows that Kahn's method for computing approximations to network behaviour can be applied to those particular synchronous networks which have been designated as modelling the processes considered by Kahn.

However, Back and Mannila do not propose, as we do here, to generalize the iterative, approximative aspect of Kahn's method beyond the level of generality considered by Kahn. On the contrary, their work, like that of Brock and Ackerman, Pratt and the relevant references on synchronous networks quoted in [1], is descriptive; the set of *all* traces of a composite process is characterized monolithically in terms of the sets of all traces of the components.

6. Conclusion

We have described a model of networks of nondeterministic processes which is a modification of the Brock–Ackerman model. For this model we have given a computational method to describe processes defined by networks.

For practical computation, the methods sketched above should be refined so as to define more closely the desired output.

For example, it would be more economical to specify, as well as nominated input sequences, the maximum number of events at each output which are of interest.

Lazy evaluation methods, as appropriate in the functional case (Kahn [3]) can also be considered, but that is beyond the scope of this note.

Appendix A. Failure of chain-completeness in Kosinski's model

In Kosinski's proposed model, the partial order $A \sqsubseteq B$ on tagged-sequence sets holds just if there is an injection $M : A \to B$ such that, for all $a \in A$, a is a prefix $M(a)$; where being a prefix requires equality of corresponding tag sets as well as of data.

Consider a countable chain of such sets,

$$Tss_1 \to Tss_2 \to \cdots$$

where we write $M_i : Tss_i \to Tss_{i+1}$ for the injection defining the ordering $Tss_i \sqsubseteq Tss_{i+1}$.

The proposal for defining a least upper bound for this chain involves considering, for each N and each $S \in Tss_N$, the set

$$\{S, M_N(S), M_{N+1}(M_N(S)), \ldots\}.$$

It forms a chain under the prefix order and so has least upper bound denoted *Ssup*.

The set of all such *Ssup* is called *Tss − sup*, and Kosinski [5] claims that it is the required least upper bound. But consider the following example, in which all tag sets are empty.

$$Tss_1 = \{0\}, \qquad Tss_2 = \{00, 010^\omega\}, \qquad Tss_3 = \{000, 010^\omega, 0010^\omega\}, \ldots,$$

$$M_1(0) = 00, \qquad M_2(00) = 000, \qquad M_2(010^\omega) = 010^\omega, \ldots.$$

Then $Tssup = \{0^\omega, 010^\omega, 0010^\omega, \ldots\}$. It is not the least upper bound as claimed in [5] because the following upper bound T is strictly less that $Tss - sup$:

$$Tssup = \{010^\omega, 0010^\omega, 00010^\omega, \ldots\}.$$

References

[1] R.J.R. Back and H. Mannila, A refinement of Kahn's semantics to handle nondeterminism and communication, *Proc. of ACM SIGACT-SIGOPS Symp. on the Principles of Distributed Computing*, Ottawa (1982) 111–120.

[2] J.D. Brock and W.B. Ackerman, Scenarios: A model of non-determinate computation, in: J. Diaz and I. Ramos, eds., *Formalisation of Programming Concepts*, Lecture Notes in Computer Science **107** (Springer, Berlin, 1981).

[3] G. Kahn, The semantics of a simple language for parallel programming, in J.L. Rosenfeld, ed., *Information Processing 74: Proc. IFIP Congress* (North-Holland, Amsterdam, 1974), pp. 471–475.

[4] R.M. Keller, Denotational models for parallel programs with indeterminate operators, in: E.J. Neuhold, ed., *Formal Description of Programming Concepts* (North-Holland, New York, 1977) pp. 337–366.

[5] P.R. Kosinski, A straightforward denotational semantics for non-determinate data flow programs, *Conf. Record 5th ACM Symp. on Principles of Programming Languages* (1978) 214–221.

[6] R. Paterson and J. Staples, An algebra of processes with a finite basis, Tech. Rept. No. 29, Department of Computer Science, University of Queensland, 1981.

[7] V.R. Pratt, A definition of 'processes', Duplicated Notes, M.I.T. Sabbat, Stanford, 1981.

[8] V.R. Pratt, On the composition of processes, *Conf. Record 9th ACM Symp. of Principles of Programming Languages* (1982) 213–223.